PRET-C: A new language for programming precision timed architectures

Sidharta Andalam, Partha Roop, Alain Girault, Claus Traulsen

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

PRET-C: A new language for programming precision timed architectures

Sidharta Andalam — Partha S Roop — Alain Girault — Claus Traulsen

N° 6922

June 4, 2009

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6922--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE – RHÔNE-ALPES

PRET-C: A new language for programming precision timed architectures

Sidharta Andalam* , Partha S Roop† , Alain Girault‡ ,
Claus Traulsen§

Thème COM — Systèmes communicants
Équipe-Projet POP ART

Rapport de recherche n° 6922 — June 4, 2009 — 35 pages

Abstract: Precision Timed Architectures (PRET) are a recent proposal for designing processors for real-time embedded systems. These processors must guarantee precise worst case reaction time (WCRT) of applications without sacrificing throughput, and must allow the WCRT of programs to be computed simply as well as efficiently. The objective of this paper is to propose a new synchronous language based on C, called PRET-C, for programming PRET machines. PRET-C supports synchronous concurrency, preemption, and a high-level construct for logical time. In contrast to existing synchronous languages, PRET-C offers C-based shared memory communications between concurrent threads, that are guaranteed to be thread safe via the proposed semantics. Preemption is also semantically simpler. Programmer can freely mix both logical time (though the notion of logical ticks) and physical time by controlling hardware timers thanks to C libraries. Mapping of logical time to physical time is achieved thanks to the WCRT analyzer and the associated compiler. We also propose the Auckland Reactive PRET processor (ARPRET) that customizes the Xilinx MicroBlaze processor, a general purpose processor (GPP). Together, PRET-C and ARPRET offer an easy, scalable, and efficient solution to the design of precision timed embedded systems.

Key-words: Real-time systems, Precision Timed (PRET) Architectures, Synchronous Languages.

* Department of ECE, University of Auckland, New Zealand sand080@aucklanduni.ac.nz
† Department of ECE, University of Auckland, New Zealand p.roop@auckland.ac.nz. Supported by research and study leave from Auckland University and a research fellowship for experienced researchers from the Alexander von Humboldt foundation.
‡ INRIA Grenoble Rhône-Alpes, POP ART project team, France, Alain.Girault@inrialpes.fr. Supported by a Marie Curie International Outgoing Fellowship within the 7th European Community Framework Programme.
§ Department of Computer Science, CAU Kiel, Germany, ctr@informatik.uni-kiel.de

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

PRET-C: Un nouveau langage pour programmer les machines temporellement prédictives

Résumé : Les architectures temporellement prédictives (PRET) ont été proposées récemment pour concevoir des processeurs pour les systèmes temps-réels embarqués. Ces processeurs doivent permettre de calculer simplement et efficacement le temps de réaction au pire cas (WCRT) des programmes, de garantir que ce WCRT est toujours respecté, sans pour autant sacrifier les performances. L'objectif de cet article est de proposer un nouveau langage de programmation synchrone basé sur C, appelé PRET-C, pour programmer les machines PRET. PRET-C supporte la concurrence synchrone, la préemption, et une construction de haut niveau pour le temps logique. Au contraire des langages synchrones existant, PRET-C offre des communications entre fils d'exécutions par mémoire partagée, dont le déterminisme est garanti grâce à la sémantique proposée. La préemption est également plus simple. Le programmeur peut combiner librement du temps logique (grâce à la notion de tick logique) et du temps physique en contrôlant des timeurs via des librairies C. La projection du temps physique sur le temps logique est obtenue grâce à l'analyseur de WCRT et au compilateur associé. Nous présentons également le processeur réactif PRET d'Auckland (ARPRET) qui customise le processeur MicroBlaze de Xilinx, un processeur général (GPP). Ensembles, PRET-C et ARPRET offrent une solution facile, efficace et qui passe à l'échelle pour la conception des systèmes embarqués temporellement prédictifs.

Mots-clés : Systèmes temps-réel, architecture temporellement prédictives (PRET), langages de programmation synchrones.

1 Introduction and Related Work

Typical embedded applications ranging from complex aircraft flight controllers to simple digital cameras require worst case guarantees on their timing performance and hence are called real-time systems. General-purpose processors, being highly speculative, are not ideally suited for implementing such systems. Though many researchers are looking at the matching of application requirements to processor technology, the design of processors with predictability, so that it directly matches the real-time applications, is still an open research problem.

Architecture support for predictable execution is not a new idea however. The Multiple Active Context System (MACS) architecture [9] was proposed in 1991. MACS is a cache-less architecture where multiple task contexts are tracked in hardware and a round robin schedule is used to issue an instruction from a different task context in each cycle. The idea of predictable architectures has gained momentum with the concept of a Precision Timed Machine (or the PRET machine) [11]. The central idea of a PRET machine is to guarantee precise timing without sacrificing throughput. Another crucial objective of a PRET machine is to simplify the worst case timing analysis of code executing on PRET machines. Since the first proposal, the UC Berkeley and Columbia group have developed a SPARC based PRET machine [15] (hereafter called the Berkeley-Columbia approach). The main idea is the introduction of a thread-interleaved pipeline executing with the support of a memory wheel so that each stage of the pipeline feeds from a new thread. In addition, they eliminate caches using scratch-pad memories and a suitable static allocation mechanism [16]. Most importantly, they have added the `deadi` instruction to achieve precise timing of segments of code: these instructions, with appropriate values of deadlines, are inserted at suitable points to ensure mutual exclusive access to shared memory.

While the Berkeley-Columbia approach is the first general purpose PRET machine executing C programs, PRET machines were also developed at Auckland University and Kiel University under the banner of reactive processors. Reactive processors, first proposed in [20], were developed with Esterel-like [4] ISA so as to eliminate the imprecise environment interaction mechanism of interrupts used in modern processors. Subsequently, they have been used to *directly execute* Esterel code both efficiently [18, 26] and precisely [5]. The precise execution is achieved using Worst Case Reaction Time analysis (WCRT) of Esterel programs to determine the tick length. Then, the processor's clock cycle is fixed to this tick length to achieve precise timing. Note that WCRT analysis of Esterel is a much simpler problem than WCET analysis of procedural languages like C, and a simple $O(n^2)$ algorithm has been developed [5]. Also, PRET architectures for Java were developed by a group in Vienna and in [21] a nice survey of predictable architectures is presented.

Although more difficult to execute predictably, C remains a language of choice for programming embedded systems: the system is split into interacting hardware and software components where the hardware components are synthesized on FPGA and the software parts are executed on an RTOS executing multithreaded C code. Yet, RTOSs add unnecessary overhead while the overall execution remains non-deterministic and hence unpredictable. The objective of the current paper is to propose simple extensions to C to ensure predictable execution on general purpose processors without the need for any RTOS. The

extensions should be such that most of them can be implemented as C-macros, eliminating the need for complex compilers. Also, the syntax and semantics of the language should be simple enough for easy adoption by C programmers. Finally, the language design should keep in mind that general purpose processor (GPPs) are essentially unpredictable, and it should facilitate easy processor customization for predictability.

Keeping these objectives in mind, we propose a set of simple synchronous extensions to C. The proposed language is called Precision Timed C (PRET-C). We use the well known synchronous semantics [2] as it offers important benefits for predictability, such as determinism and reactivity. Our extensions to C are the following ones: `ReactiveInput`, `ReactiveOutput`, `EOT`, `PAR`, and `abort when pre`. The `ReactiveInput` and `ReactiveOutput` statements declare the environment inputs and outputs. The `EOT` construct ends the local tick of a thread. The global tick of the program occurs when all active threads have completed their local ticks. Thus, `EOT` offers the programmer a very simple scheme of managing logical time. The `PAR` construct is used for creating concurrent threads, where all threads have fixed priority based on their syntactic order inside the `PAR`. Threads communicate using C shared variables. Threads are automatically thread safe due to the sequential semantics of the `PAR` construct along with the atomicity of read and write operations. The `abort when pre` construct preempts a thread when a specified condition is satisfied.

Synchronous extensions to C are not new [6, 7, 13]. ReactiveC [6] is closest to our approach and the `PAR` construct is similar to the `par` in ReactiveC. In contrast, we allow multiple readers of a shared variable to read different values of this variable during an instant. ReactiveC, on the other hand, requires a stricter notion of data-coherency, where all readers must read the same value of data. We relax this restriction and hence PRET-C programs can have unrestricted access to shared data, but the overall program remains deterministic due to the semantics of the language. This facilitates easy management of shared resources by the programmer without the need for complex OS-based synchronization features. ECL [13] is a more recent synchronous C extension that is closer to Esterel, and its synchronous parallel construct is exactly like Esterel. The main means of communication between threads in ECL is through signals, like in Esterel. Compared to PRET-C all these languages are more complex and were not designed with predictability in mind. More recently, a language called SC [23] has been proposed for encoding Statecharts in C. However, compared to PRET-C it is more complex due to the need for enforcing the semantics of Statecharts.

Concerning the architecture, we propose a PRET architecture called ARPRET (Auckland Reactive PRET) by customizing an existing soft-core GPP. ARPRET is inspired by reactive processors [20, 26, 14] and the PRET machine of [11]. However, it enhances and extends both. Firstly, unlike the custom designed PRET machines, we propose simple hardware extensions to soft-core embedded processors. Our hardware extension is demonstrated by extending the MicroBlaze soft-core processor from Xilinx [24]. However, this extension is feasible on any other soft-core processor. Hence, our proposal achieves PRET implementation by using significantly less hardware resources and demonstrates the reuse of soft-core processor for PRET.

While ARPRET is a reactive processor [26], it significantly extends all earlier reactive processors. Firstly, all existing reactive processors are capable of exe-

cuting only *pure Esterel* without any C function calls. Secondly, earlier reactive processors rely on a WCRT analysis that has been shown to be pessimistic [5]. Finally, all earlier reactive processors are custom processors. We illustrate that by simple customizations of GPPs and using C-macros, we can design PRET machines that use very minimal hardware customizations.

The organization of this paper is as follows. In Section 2 we present the PRET-C language through a producer consumer example along with an intermediate format called TCCFG. This is followed by the semantics in Section 3. In Section 4 we present the ARPRET architecture and how PRET-C programs are executed. The experimental results are presented in 6 and conclusions are presented in 7.

2 PRET-C overview

The overall design philosophy of PRET-C and the associated architecture may be summarized using the following three simple concepts:

- Concurrency: Concurrency is logical but execution is sequential. This is used to ensure both synchronous execution and thread-safe shared memory communication. This has been the founding principle of the synchronous programming languages [2].
- Time: Time is logical and the mapping of logical time to physical time is achieved by the compiler and the WCRT analyzer [19].
- Design approach: ARPRET achieves PRET by simple customizations of GPPs. The extensions to the C are minimal are implemented through C-macros.

2.1 PRET-C language extensions

Statement	Meaning
`ReactiveInput I`	declares `I` as a reactive input coming from the environment
`ReactiveOutput O`	declares `O` as a reactive output emitted to the environment
`PAR(T1,...,Tn)`	synchronously executes in parallel the `n` threads `Ti`, with higher priority of `Ti` over `Ti+1`
`EOT`	marks the end of a tick (local or global depending on its position)
`[weak] abort P when pre C`	immediately kills `P` when `C` is true in the previous instant

Table 1: PRET-C extensions to C.

PRET-C (Precision Timed C) is a *synchronous* extension of the C language similar in spirit to ECL and ReactiveC. Unlike the earlier synchronous C extensions, it is based on a *minimal* set of extensions and is specially designed for *predictable* execution on ARPRET. It extends C using the five constructs shown in Table 1. In order to guarantee a predictable execution, we impose the following four restrictions on the C language:

- Pointers and dynamic memory allocation are disallowed to prevent unpredictability of memory allocation.
- All loops must have at least one `EOT` in their body. This is needed to ensure that thread compositions are deadlock free. This restriction could be relaxed for the loops that can be statically proven to be *finite*.
- All function calls have to be non-recursive to ensure predictable execution of functions.
- Jumps via `goto` statements are not allowed to cross logical instants (i.e., `EOT`s).

Our five C extensions are implemented as C-macros, all contained in a `pretc.h` file that must be included at the beginning of all PRET-C programs. As a result, we only rely on the `gcc` macro-expander and compiler for compiling PRET-C programs.

Like any C program, a PRET-C program starts with a preamble part (`#define` and `#include` lines), followed by global declarations (reactive inputs, reactive outputs, and classical C global variables), and finally function definitions (including the `main` function).

A PRET-C program runs periodically in a sequence of *ticks* triggered by an external clock. The inputs coming from the environment are sampled at the beginning of each tick. They are declared with the `ReactiveInput` statement. The outputs emitted to the environment are declared with the `ReactiveOutput` statement. Reactive inputs are read from the environment at the beginning of every tick and cannot be modified inside the program. Hence, the value of these variables remain fixed throughout an instant. They have a default value when the environment assigns no value. In contrast, reactive outputs may be updated by the program and can have several values within an instant. The final value of these variables (termed their steady-state value) is emitted to the environment. Reactive outputs behave exactly like normal variables in C, except that they are emitted to the environment while normal variables are for communication between threads and are not seen in the environment.

Programmers familiar with usual synchronous languages such as Esterel [4] or its earlier C-based extensions [13] will notice the difference with PRET-C. Unlike these languages where the primary means of thread communication is *signals*, we use *variables*. Signals have both *status* and an associated *value* when the signal is not pure. Esterel forbids the usage of variables for communication between threads for causality reasons. PRET-C allows unrestricted shared variable access across threads and thread safe communication is achieved using the static thread priority and the semantics of our parallel operator (see Section 3). Besides this, our reactive inputs and outputs are similar to Esterel's input and output signals, respectively.

The `PAR(T1,...,Tn)` statement spawns `n` threads that are executed in lock step. All spawned threads evolve based on the same view of the environment. However, unlike the usual ‖ of other synchronous languages like Esterel, where threads are scheduled in each instant based on their signal dependencies, threads in PRET-C are always scheduled based on a fixed static order. This is determined based on the order in which threads are spawned using the `PAR` construct. E.g., a `PAR(T1,T2)` statement assigns to `T1` a higher priority than to `T2`.

Parallel threads communicate through shared variables and reactive outputs. The task of ensuring mutually exclusive access is achieved by ensuring that, in every instant, all threads are executed in a fixed total order by the scheduler. When more than one thread acts as a writer for the same variable, then the execution semantics of the program still remains deterministic. Indeed, race conditions can happen in RTOSs when these writes are non-atomic, i.e, if one write operation can be interrupted and another thread can then modify the same variable. However, as long as these writes happen atomically in some fixed order, the value of the result will be always predictable and race conditions will be prevented. This is ensured by our ARPRET architecture thanks to the proposed multi-threaded execution. On ARPRET, once a thread starts its execution, it cannot be interrupted. The next thread is scheduled only when the previous thread reaches its `EOT`. Thus, when two or more threads can modify the same variable, they always do so in some fixed order, ensuring that the data is consistent.

The `EOT` statement marks the end of a tick. When used within several parallel threads, it implements a *synchronization barrier* between those threads. Indeed, each `EOT` marks the end of the *local tick* of its thread. A *global tick* elapses only when all participating threads of a `PAR()` reach their respective `EOT`. In this sense, `EOT` is similar to the `pause` statement of Esterel. `EOT` enforces the synchronization between the parallel threads by ensuring that the next tick is started only when all threads have reached their `EOT`. Finally, it allows to compute precisely the WCRT of a program by computing the required execution time of all the computations scheduled between any two successive `EOT`. This WCRT analysis is presented in a companion report [19].

`EOT` is similar in spirit to the `deadi` instruction of [15] (immediate deadline). However, unlike the *low-level* `deadi` instruction that manages timing by associating timers, `EOT` is a *high-level* programming construct. The task of ensuring precise timing of threads is not left to the programmer but is derived by WCRT analysis and is a compilation task [19]. Moreover, the deadline instruction is also used for achieving mutual exclusion by time-interleaving the access to shared memory. This is achieved by setting precise values to the deadlines. However, if done manually, this task can be very complex, even for simple programs. This is mainly due to arbitrary branching constructs and loops. Automating this task is non-trivial and has not been solved in [15]. Our solution to achieve mutual exclusive access to shared memory, on the other hand, is ensured by having static thread priorities, and then scheduling the threads in this fixed linear order in every instant.

The `abort P when pre C` construct preempts its body `P` immediately when the condition `C` is true (like `immediate abort` in Esterel). Like in Esterel, preemption can be either *strong* (`abort` alone) or *weak* (when the optional `weak` keyword is used). In case of a strong abort, the preemption happens at the beginning of an instant, while the weak abort allows its body to execute

and then the preemption triggers at the end of the instant. All preemptions are triggered by the *previous* value of the Boolean condition (hence the `pre` keyword), to ensure that computations are deterministic. This is needed since the values of variables can change during an instant. The use of the `pre` ensures that preemptions are always taken based on the steady state values of variables from the previous instant. In other words, like in ReactiveC, we use a restricted form of causality compared to Esterel.

2.2 A Producer Consumer example

We present a producer-consumer adapted from [22] to motivate PRET-C. It is shown in Listing 1. The program starts by including the `pretc.h` file (line 1). Then, reactive inputs are declared (lines 3 and 4), followed by regular global C variables (lines 5 and 6), and finally all the C functions are defined (lines 7 to 41). The `main` function consists of a single main thread that spawns two threads (line 35): a `sampler` thread that reads some data from the `sensor` reactive input and deposits this data on a global circular `buffer`, and a `display` thread that reads the deposited data from `buffer` and displays this data on the screen, thanks to the user defined function `WriteLCD` (line 29). The `sampler` and `display` threads communicate using the shared variables `cnt` and `buffer`. Also, the programmer has assigned to the `sampler` thread a higher priority than to the `display` thread.

During its first local tick, the `sampler` thread does nothing. During its second local tick, it checks if its data `buffer` is full (line 11): as long as `buffer` is full, it keeps on waiting until the display thread has read some data so that there is empty space in `buffer`. When it exits this while loop, then it writes the current instant's value of the `sensor` input to the next available location of the buffer (line 12) and ends its local tick (line 13). In its last local tick, the `i` index of the buffer and the total number of data `cnt` in the buffer are incremented (lines 14 and 15), since this is a circular buffer. Then the sampling loop is restarted.

During its first local tick, the `display` thread checks if there is any data available to read from the buffer (line 23). If there is no data available, then it ends its local tick and keeps on waiting until some data is deposited by the producer. When this happens, it reads the next data from `buffer` (line 24) and ends its local tick (line 25). In its next local tick, the `i` index of the buffer is incremented (line 26) and the total number of data `cnt` in the buffer is decremented (line 27). During its last local tick, it sends the data read from the buffer to a display device (line 29).

The main thread (main function) has an enclosing `abort` over the `PAR` construct. This preemption is taken whenever an external `reset` button has been pressed in the previous instant (line 36). In our example, when a strong preemption happens, the two threads are aborted and the program flushes the `buffer` (line 38), initializes the `cnt` variable (line 37), and pauses for an instant before restarting the two threads again.

The execution of this code on any GPP with an RTOS to emulate concurrency will lead to race conditions. It is the responsibility of the programmer to ensure that critical sections are properly implemented using OS primitives such as semaphores. This will happen because of non-exclusive accesses to the shared

Listing 1: A Producer Consumer in PRET-C

```
#include <pretc.h>
#define N 1000
ReactiveInput (int, reset, 0);
ReactiveInput (float, sensor, 0.0);
int cnt=0;
float buffer[N];
void sampler() {
  int i=0;
  while(1) {
    EOT;
    while (cnt==N) EOT;
    buffer[i]=sensor;
    EOT;
    i=(i+1)%N
    cnt=cnt+1;
  }
}
void display() {
   int i=0;
   float out;
   initLCD();
   while(1) {
     while (cnt==0) EOT;
     out=buffer[i];
     EOT;
     i=(i+1)%N;
     cnt=cnt-1;
     EOT;
     WriteLCD(out);
   }
}
void main() {
  while{1} {
    abort
      PAR(sampler,display);
    when pre (reset);
    cnt=0;
    flush(buffer);
    EOT;
  }
}
```

variable `cnt`. However, on ARPRET, the execution will be always deterministic. Assume that `cnt=cnt+1` and `cnt=cnt-1` happen during the same tick. Due to the higher priority of `sampler` over `display`, `cnt` will be first incremented by 1, and once `sampler` reaches its `EOT`, the ARPRET scheduler (see Section 4) will select `display` which will then decrement `cnt` by 1. Thus, the value of `cnt` will be consistent without the need for enforcing mutual exclusion between the `sampler` and `display` threads.

2.3 TCCFG intermediate format

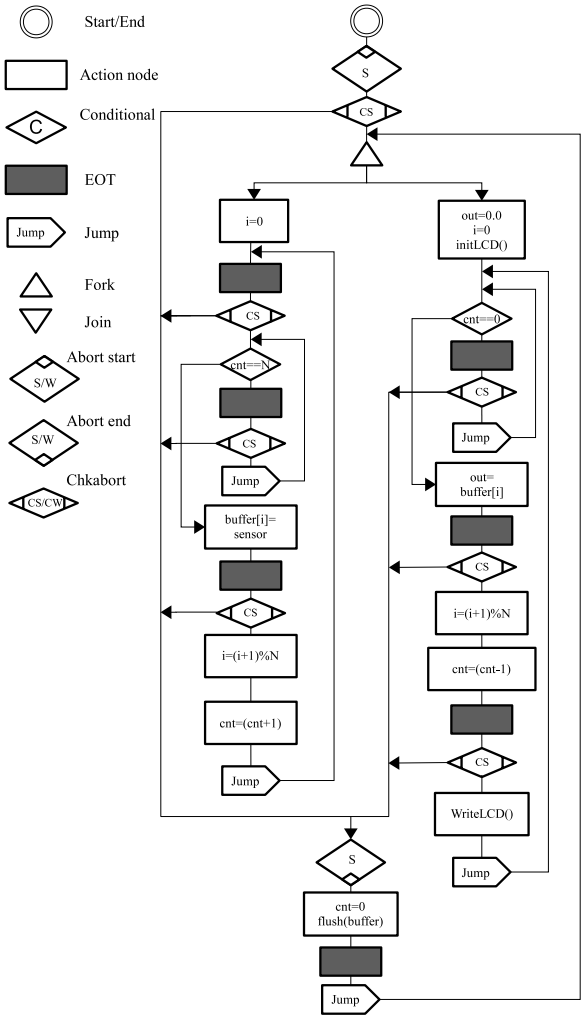


Figure 1: TCCFG of the Producer-Consumer

We propose a new intermediate format for PRET-C programs, called Timed Concurrent Control Flow Graph (TCCFG). TCCFG is a control flow graph similar in spirit to the CCFG of [12]. However, it is much simpler than earlier

intermediate formats for Esterel because our language is simpler. The TCCFG corresponding to our example of Listing 1 is shown in the Figure 1.

A PRET-C program is first converted into ARPRET assembly code, and then the TCCFG is automatically extracted from this assembly code. We generate the intermediate format from the assembly code (rather than from the source code) so as to get, for each tick, precise values in terms of ARPRET clock cycles. Moreover, by working on the assembly code, compiler optimizations need not be turned off. We use TCCFG for WCRT analysis of PRET-C, which is presented in a companion report [19].

The TCCFG encodes the explicit control-flow of the threads as well as the forking and joining information of the threads. It has the following types of nodes:

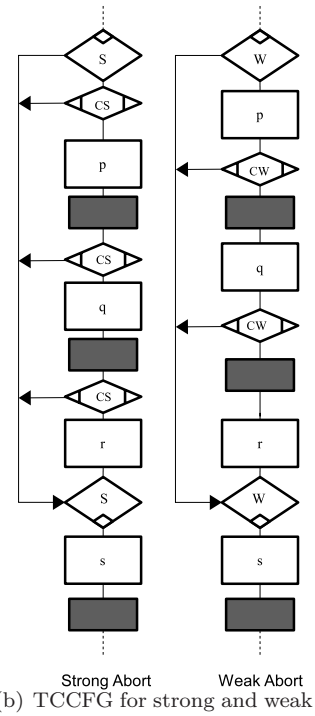
- Start/End node: Every TCCFG has a unique start node where the control begins, and may have an end node if the program can terminate. Both are drawn as concentric circles.
- Fork/Join nodes: They mark and where concurrent threads of control start and end. They are drawn as triangles.
- Action nodes: They are used for any C function call or data computation. We use rectangles to draw them.
- EOT nodes: They indicate the end of a local or a global tick, and are drawn as filled rectangles.
- Control flow nodes: We have two types of control flow nodes: conditional nodes to implement conditional branching (drawn as rhombus labelled with the condition) and jump nodes for mapping unconditional branches (drawn as arrow-shaped pentagons) which are needed for infinite loops.
- Abort nodes: We have abort start and abort end nodes to mark the scope of an abort. They are drawn as rhombuses labeled either with '`s`' or '`w`' to indicate strong or weak aborts.
- Checkabort nodes: These are special nodes that implement the semantics of aborts. They are drawn as thin rhombuses labeled either with '`cs`' or '`cw`' to indicate strong or weak checkaborts.

Depending on the type of aborts, we insert checkabort nodes at precise points using the following structural translation rules. This is illustrated in Figure 2:

1. For each strong abort in the program, a checkabort node is inserted *just after* every EOT node to check for the preemption condition at the beginning of every tick. Moreover, a checkabort node is inserted at the beginning of the body of the enclosing thread of this abort, to instantly trigger the preemption without executing this body (as per the semantics of strong preemptions in PRET-C). PRET-C's strong aborts correspond to Esterel's strong immediate preemptions.
2. For each weak abort in the program, we insert a checkabort node *just before* every EOT node. PRET-C's weak aborts correspond to Esterel's weak immediate preemptions.

```
0  //Strong Abort Example
1  abort
2          p;
3          EOT;
4          q;
5          EOT
6           r;
7  when pre(C);
8  s;
9  EOT;
10 //Weak Abort Example
11 weak abort
12         p;
13         EOT;
14         q;
15         EOT
16          r;
17 when pre(C);
18 s;
19 EOT;
```

(a) Strong and weak abort in PRET-C.



(b) TCCFG for strong and weak aborts.

Figure 2: (a) Structural translation of aborts sample code; (b) TCCFG.

3 Semantics of PRET-C

In this section we present the semantics of PRET-C using structural operational style [17]. Our semantics resembles the semantics of earlier synchronous languages like Esterel [3] and the SL language [8]. The main difference from earlier synchronous semantics is due to the encoding of parallel as a fixed sequence based on the priority of threads. We use this to ensure *determinism*s of all PRET-C programs with arbitrary data dependencies. Another key difference is the way we handle preemptions. Unlike earlier approaches that depend on the translation of aborts to traps and then defining the semantics based on traps, we define the semantics of aborts directly. Due to the translation of aborts to traps, weak aborts are encoded using extra concurrent threads within the trap body. For dealing with strong aborts, additional suspension is needed. Unlike this, we propose a kernel statement called *checkabort*. We then offer a structural translation that inserts *checkabort* terms at appropriate points in code (exactly like the insertions of `checkabort` nodes as described in Section 2.3). As a consequence, we can treat aborts semantically without the need for introducing extra threads during the translation. We start by presenting a set of structural translation rules for inserting *checkabort* statements.

3.1 Structural translation rules for preemption

To be able to deal with preemptions, we need a kernel statement called *checkabort* in addition to the preemption statement itself. The *checkabort* statement is

needed to check if the preemption condition is satisfied. Also, this checking has to be done at specific points in the code to preserve the semantics of strong versus weak preemption in PRET-C.

The first step in the semantics is to structurally translate abort bodies such that appropriate *checkabort* statements are inserted within the body. The rules for inserting these statements is as follows (and is identical to the rules for inserting `checkabort` nodes in TCCFG as outlined in Section 2.3).

1. For every strong abort, a *checkabort* statement is inserted after every EOT to check for the preemption condition at the beginning of every instant. Also a *checkabort* statement is inserted at the beginning of the body to immediately trigger the preemption without executing the body (as per the semantics of strong preemptions in PRET-C that behave like strong immediate preemptions in Esterel).

2. For every weak abort, we insert a *checkabort* statement immediately before every `EOT`. Weak aborts correspond to weak immediate abortion in Esterel.

3. When abort statements are nested, this explicitly captures the priority. Like Esterel, the outer abort statements have higher priority over the inner ones. Also, to capture the behaviour of strong aborts and weak aborts properly, the strong aborts need to be checked from outer to inner aborts while the weak aborts need to be checked from the inner to outer. We insert check abort statements such that we check for strong aborts at the beginning of every instance from the outermost to the innermost strong abort. For weak aborts, we check at the end of every instance from the innermost to the outermost (to take care of the chaining of nested weak abort bodies).

The structural translation approach is illustrated by the following sample code in Listing 3.1 and its translated code with appropriate *checkabort* statements inserted in Listing 3.1. The *checkabort* statements are inserted as per the structural translation rules. Priority is explicitly encoded by nesting of *checkabort* statements. For example, on line 16 and 17 we first check for the outermost strong abort (condition c4) followed by the inner strong abort (condition c2). Similarly on lines 21 and 22, we first check the inner weak abort (condition c1) followed by the outer weak abort (condition c3).

3.2 Kernel language and operational rules

We start by presenting the *kernel language*, summarized in Table 2. We do not use this kernel for compiling, only to express the semantics of the language. One important point is that, in a PRET-C program, an `EOT` marks the end of the local tick of a thread whenever it is inside the scope of a `PAR`, or the end of the global tick otherwise.

Then, any program transition is represented by an SOS rule of the form $E : t \xrightarrow[I]{k} E' : t'$ where:

- t represents a term that consists of any arbitrary composition of kernel statements.

```
abort
      p;
      EOT;
      weak abort
        EOT;
        q;
        abort
             r;
             EOT;
             weak abort
                  while(1){
                      s;EOT;
                     ||
                      u;EOT;v;
                  }
             when pre c1;
             A1:x;
        when pre c2;
        A2:y;
        EOT;
      when pre c3;
when pre c4;
A3/A4: z;
EOT;
```

(a) NestedAborts.c: A Sample program with Nested aborts in PRET-C

```
abort
      checkabort(pre(c4),A4);
      p;
      EOT;
      checkabort(pre(c4),A4);
      weak abort
        checkabort(pre(c3),A3);
        EOT;
        checkabort(pre(c4),A4);
        q;
        abort
             checkabort(pre(c2),A2);
             r;
             checkabort(pre(c3),A3);
             EOT;
             checkabort(pre(c4),A4);
             checkabort(pre(c2),A2);
             weak abort
                  while(1){
                      s;
                      checkabort(pre(c1),A1);
                      checkabort(pre(c3),A3);
                      EOT;
                      checkabort(pre(c4),A4);
                      checkabort(pre(c2),A2);
                     ||
                      u;
                      checkabort(pre(c1),A1);
                      checkabort(pre(c3),A3);
                      EOT;
                      checkabort(pre(c4),A4);
                      checkabort(pre(c2),A2);
                      v;
                  }
             when pre c1;
             A1:x;
        when pre c2;
        A2:y;
        checkabort(pre(c3),A3);
        EOT;
        checkabort(pre(c4),A4);
      when pre c3;
when pre c4;
A3/A4: z;
EOT;
```

(b) Structural translation of NestedAborts.c

Figure 3: Nested aborts and their structural translation

Statement	Meaning
nop	terminates instantaneously without doing anything
EOT	marks the completion of a reaction (global tick) when all local ticks have been reached
$t; u$	sequential the execution of t followed by u
$PAR(t, u)$	logical parallel execution of t and u with higher priority for t
$v = f(...)$	computes f and then assigns the result to variable v
$if(c)\{t\}else\{u\}$	conditional statement
$while(1)\{t\}$	infinite loop where t must have at least one EOT
$while(c)\{t\}$	possibly finite loop where t must have at least one EOT
$checkabort(cond, label)$	returns the value of $label$ when the preemption condition $cond$ is true
$[weak]\, abort\, P\, when\, pre\, c$	weak or strong preemption of P when the condition $pre\, c$ is true

Table 2: Kernel statements of PRET-C.

- t' represents the residual of a term after the transition is taken.
- I represents the set of reactive inputs.
- E represents the status or valuations of a set of variables (both global and local).
- E' represents the status of the same set of variables after the transition has been taken.
- k denotes the completion code after the transition has been taken. Transition may complete with code 0 to represent that a nop statement has been executed, 1 to represent that an EOT has been executed, a label to represent that a preemption condition is true, and $\perp$ to represent that any transition other than the above has been executed. The environment E will change to some E' only when the completion code of the rule in question is $\perp$. These are needed for proper execution of the PAR statement. We use the partial order $0 < 1 < label$ on completion codes, hence $k > 1$ states that k is a label. Concretely, the label is the continuation address after the preemption is performed. This encoding of completion codes is inspired (although simpler) by the encoding of the trap codes in Esterel's semantics.

The `nop` statement: This statement terminates instantaneously without modifying the variables:

$$E : nop \xrightarrow[I]{0} E : nop \tag{1}$$

The `EOT` statement: The execution of the EOT statement completes the local tick of the program and this is captured by a completion code of 1:

$$E : EOT \xrightarrow[I]{1} E : nop \tag{2}$$

The sequence operator: The right branch u is executed sequentially after the left branch t:

If t is any statement other than *nop*, EOT and t does not trigger an preemption then this rule states that, after the term t becomes t' due to the execution of a statement in the LHS, the term t' will be in sequence with u. The status of the variables will change from E to E' due to the execution of the current micro-step step. Since the current step executed a statement that is neither EOT nor *nop*, the completion code is $\perp$.

$$\frac{E : t \xrightarrow[I]{\perp} E' : t'}{E : t; u \xrightarrow[I]{\perp} E' : t'; u} \tag{3}$$

This rule asserts the fact that when t pauses (by executing an EOT), the sequence also pauses.

$$\frac{E : t \xrightarrow[I]{1} E : t'}{E : t; u \xrightarrow[I]{1} E : t'; u} \tag{4}$$

This rule captures the situation when t terminates. The control is then immediately passed to u:

$$\frac{E : t \xrightarrow[I]{0} E : nop}{E : t; u \xrightarrow[I]{0} E : u} \tag{5}$$

This rule captures the execution of a *checkabort* term t that triggers an enclosing preemption. In this event, the sequence terminates with a completion code that is equal to the value returned by *checkabort*.

$$\frac{E : t \xrightarrow[I]{k} E : nop}{E : t; u \xrightarrow[I]{k} E : nop}(k > 1) \tag{6}$$

The Assignment statement: A variable can be assigned a value returned by a function or the result of the evaluation of an expression (which can be also encapsulated as a function). This rule states that when such a statement is executed, the value of the variable v gets updated by the return value of the function in the set of variables E.

$$\frac{e_1..e_i \in E, i_1..i_j \in I}{E : v = f(e_1..e_i, i_1..i_j) \xrightarrow[I]{\perp} E' : nop}(E' = E[v \leftarrow f(e_1..e_i, i_1..i_j)]) \tag{7}$$

The conditional statement: There are two rules corresponding to the case when the guard condition of the conditional statement is true and false respectively.

When the guard condition is true, this rule states that the if branch (the term t) gets executed next.

$$\frac{e_1..e_i \in E, i_1..i_j \in I, c(e_1..e_i, i_1..i_j) = tt}{E : \mathit{if}\,(c(e_1..e_i, i_1..i_j))\,\{t\}\;\mathit{else}\,\{u\} \xrightarrow[I]{\perp} E : t} \tag{8}$$

When the guard condition is false, this rule states that the else branch (the term u) gets executed next.

$$\frac{e_1..e_i \in E, i_1..i_j \in I, c(e_1..e_i, i_1..i_j) = ff}{E : \mathit{if}\,(c(e_1..e_i, , i_1..i_j))\,\{t\}\;\mathit{else}\,\{u\} \xrightarrow[I]{\perp} E : u} \tag{9}$$

Infinite Loop: An infinite loop is rewritten as a sequence consisting of the body followed by the loop itself. We further demand in our semantics that t executes at least on `EOT` for every input to avoid the rewriting of the loop into an infinite sequence of unfinished transitions.

$$E : \mathit{while}(1)\,\{t\} \xrightarrow[I]{\perp} E : t; \mathit{while}(1)\,\{t\} \tag{10}$$

Finite Loop: There are two cases depending on whether the guard condition is true or false. We demand that the loop body t must have at least one `EOT` to prevent deadlocks (due to dependency of the loop condition with other threads).

When the guard condition is true, the body of the while will execute once. The completion code will be a value k based on the completion code of the body and the status of the variables will get updated from E to E' due to the execution of the body once. If the term t reaches an `EOT`, then the while terminates with a completion code of 1. If the body finishes then the term t finishes with a completion code of 0 and the while condition in sequence is again evaluated.

$$\frac{e_1..e_i \in E, i_1..i_j \in I, c(e_1..e_i, i_1..i_j) = tt}{E : \mathit{while}(c(e_1..e_i, i_1..i_j))\,\{t\} \xrightarrow[I]{\perp} E : t; \mathit{while}(c(e_1..e_i, i_1..i_j))\,\{t\}} \tag{11}$$

When the guard condition is false, the statement simply terminates.

$$\frac{e_1..e_i \in E, i_1..i_j \in I, c(e_1..e_i, i_1..i_j) = ff}{E : \mathit{while}(c(e_1..e_i, i_1..i_j))\,\{t\} \xrightarrow[I]{\perp} E : \mathit{nop}} \tag{12}$$

The PAR **statement**: The parallel execution of the terms t and u, where the priority of t is higher than that of u, is captured by first executing all statements of t until the local tick of t is reached (`EOT` statement). This is captured by the rule 13 (that reduces t until the EOT) and 14 (that pauses t until u reaches its EOT) respectively.

$$\frac{E : t \xrightarrow[I]{\perp} E' : t'}{E : PAR(t, u) \xrightarrow[I]{\perp} E' : PAR(t', u)} \tag{13}$$

$$\frac{E : t \xrightarrow[I]{1} E : t' \quad E : u \xrightarrow[I]{\perp} E' : u'}{E : PAR(t,u) \xrightarrow[I]{\perp} E' : PAR(t,u')} \tag{14}$$

The PAR pauses with a completion code of 1 when both threads have reached their EOT (rule 15). Alternatively, if t is at an EOT while u terminates, the PAR also terminates with a completion code of 1 (rule 16). In both cases, the PAR has reached the end of its tick; whether this is a global or a local tick depends on the nesting of PARs.

$$\frac{E : t \xrightarrow[I]{1} E : t' \quad E : u \xrightarrow[I]{1} E : u'}{E : PAR(t,u) \xrightarrow[I]{1} E : PAR(t',u')} \tag{15}$$

$$\frac{E : t \xrightarrow[I]{1} E : t' \quad E : u \xrightarrow[I]{0} E : nop}{E : PAR(t,u) \xrightarrow[I]{1} E : PAR(t',nop)} \tag{16}$$

When t terminates, PAR just becomes u, just like rule 5 for sequence (rule 17). The asymmetry between the rules 16 (when u terminates) and rule 17 (when t terminates) is due to the priority of t over u.

$$\frac{E : t \xrightarrow[I]{0} E : nop}{E : PAR(t,u) \xrightarrow[I]{0} E : u} \tag{17}$$

The next two deriving rules are needed to handle preemption of the PAR due to any enclosing preemption statements. Rule 18 is needed when the first thread has reached a *checkabort* statement such that the execution of this statement will return a completion code $k > 1$. In this event, the first thread pauses and the execution of the second thread starts. The execution of the second thread will then continue until its matching *checkabort*.

$$\frac{E : t \xrightarrow[I]{k} E : t' \quad E : u \xrightarrow[I]{\perp} E' : u'}{E : PAR(t,u) \xrightarrow[I]{\perp} E' : PAR(t,u')}(k > 1) \tag{18}$$

The PAR exits with a completion of of $k > 1$ when both threads exit with the same completion code. This happens when both threads exit by executing their respective *checkabort* statements in response to an enclosing preemption. This is captured by rule 19 below.

$$\frac{E : t \xrightarrow[I]{k} E : nop \quad E : u \xrightarrow[I]{k} E : nop}{E : PAR(t,u) \xrightarrow[I]{k} E : nop}(k > 1) \tag{19}$$

The behavior expressed in rules 18 and 19 for preemption is similar to that of rules 14 and 15 for EOT.

3.3 Dealing with Preemption

The *checkabort* **statement**: When the preemption condition is true, *checkabort* returns a completion code that is equal to the continuation address (A). Note that the preemption condition is defined over the variable statuses in the previous instance (E^p and I^p).

If the preemption condition is true, then the *checkabort* statement is rewritten as a *nop* and returns a value > 1.

$$\frac{e_1..e_i \in E^p, i_1..i_j \in I^p, c(e_1..e_i, i_1..i_j) = tt}{E : checkabort(c(e_1..e_i, i_1..i_j), A) \xrightarrow[I]{A} E : nop} \tag{20}$$

If the preemption condition is false, on the other hand, then the *checkabort* statement terminates without doing any thing.

$$\frac{e_1..e_i \in E^p, i_1..i_j \in I^p, c(e_1..e_i, i_1..i_j) = ff}{E : checkabort(c(e_1..e_i, i_1..i_j), A) \xrightarrow[I]{\perp} E : nop} \tag{21}$$

The Abort and the Weak Abort Statements: The rules for both strong and weak preemption is the same. The actual distinction between the behaviour of the two is achieved by the structural translation rules presented earlier.

When the body executes a sequence of instantaneous transitions or reaches an EOT such that the preemption condition is not satisfied, then rules 22 and 23 are applicable respectively. In these cases, the body is simply replaced by the consequent term. This is captured by the following two rules.

$$\frac{E : t \xrightarrow[I]{\perp} E' : t'}{E : [weak]\, abort\, t\, when\, pre\, c \xrightarrow[I]{\perp} E' : [weak]\, abort\, t'\, when\, pre\, c} \tag{22}$$

$$\frac{E : t \xrightarrow[I]{1} E : t'}{E : [weak]\, abort\, t\, when\, pre\, c \xrightarrow[I]{1} E : [weak]\, abort\, t'\, when\, pre\, c} \tag{23}$$

The body may execute a *checkabort* statement such that the termination code is > 1. In this event, we have to distinguish between two cases as shown in the rules 24 and 25 respectively. Rule 24 captures the fact that the abortion condition, checked by the *checkabort* term that returned a label k, matches the condition of the abort statement. In this case, the abort is simply rewritten as *nop*. Since the abortion was taken, the abort statement returns $\perp$.

The second case, as captured by the rule 25, is needed when the *checkabort* corresponds to an enclosing abort and hence, the inner abort is rewritten as a *nop* to indicate that the abort was not executed. Unlike rule 24, rule 25 returns k to its enclosing context.

$$\frac{E : t \xrightarrow[I]{k} E : nop, k > 1}{E : [weak]\, abort\, t\, when\, pre\, c \xrightarrow[I]{\perp} E : nop}(t = checkabort(c, k); t') \tag{24}$$

```
abort
   abort
       EOT;
       checkabort(c2, A2);
       checkabort(c1, A1);
       t;u;
   when pre c1;
   A1:
when pre c2;
A2:
```

Figure 4: Example of preemption

$$\frac{E : t \xrightarrow[I]{k} E : nop, k > 1}{E : [weak]\, abort\, t\, when\, pre\, c; label \xrightarrow[I]{k} E : nop}(t = checkabort(c1, k); t') \quad (25)$$

In order to illustrate these two cases, we consider the program shown in Figure 4. Upon the preemption, the outermost `checkabort` (resp. innermost) will return as its completion code the address of the label `A2` (resp. `A1`). If the outermost `checkabort` returns `A2`, then the innermost `checkabort` need not be executed, and hence is rewritten as *nop* by the rule . Also, this abort statement will return `A2` to its environment (i.e., the enclosing abort). If, on the other hand, the outermost `checkabort` returns *nop* and the innermost one returns `A1`, then using rule , the inner abort will trigger and will transfer control to `A1`.

The last case happens when the body terminates normally. In this event, the abort statement is replaced by nop.

$$\frac{E : t \xrightarrow[I]{0} E : nop}{E : [weak]\, abort\, t\, when\, pre(c) \xrightarrow[I]{0} E : nop} \quad (26)$$

Note that the actual branching to outside the body of an abort is performed by the abort statement and not by the *checkabort*. This is necessary to handle the execution of parallel *checkabort* correctly: even if the preemption is triggered in multiple sub-threads, we only want to perform the jump once. In particular, we cannot replace the *checkabort* by a conditional jump.

3.4 Illustration

Consider the following PRET-C program where $t1 \cdots t9$ are terms devoid of any EOT terms:

```
weak abort
PAR(// first thread
    PAR(t1, t2);
    PAR(t3; EOT, t4; EOT);
    EOT; t7,
```

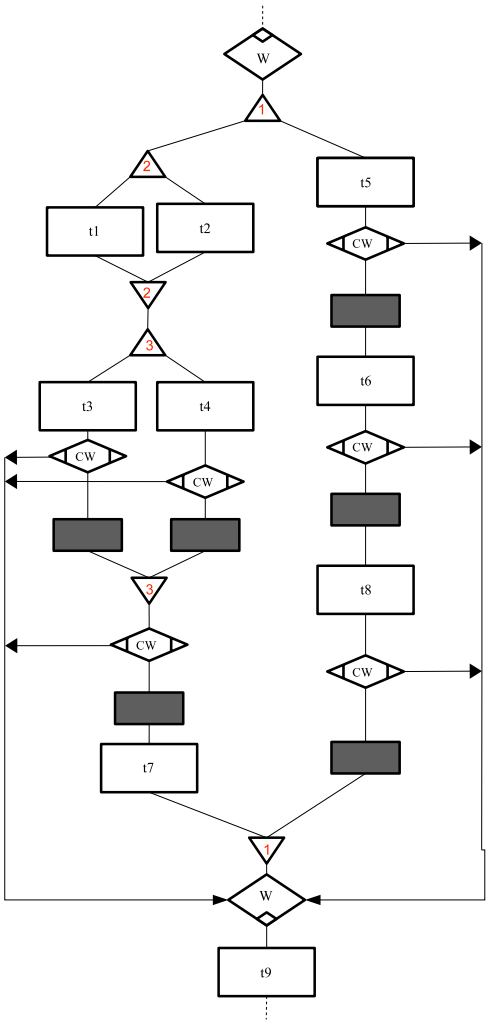


Figure 5: TCCFG example for illustrating the semantics

```
    // second thread
    t5; EOT; t6; EOT; t8; EOT
   )
when pre I;
END: t9
```

The scheduling and synchronization between threads of this program is explained by the TCCFG as shown in Figure 5. In the following we will use the notation *PAR*-i to refer to the i-th *PAR*. We will consider the following two execution scenarios.

- I is *absent* in the previous instant: Then *PAR*-1 will start and will start *PAR*-2 immediately. *PAR*-2 by rule 13 will start $t1$ that will execute and eventually become a *nop*. Then, by rule 17, *PAR*-2 just becomes $t2$ and eventually terminates when $t2$ terminates. Then, *PAR*-3 starts

and by rule 13 $t3$ is executed until its EOT. When this happens, due to the rule 14, $t3$ pauses and the execution of $t4$ starts. Finally, PAR-3 returns a completion code of 1, using rule 15 when $t4$ also reaches its EOT. Now, PAR-1 starts the execution of the second thread $t5$, using rule 14. Finally, when $t5$ reaches its EOT, the global tick happens by rule 15. The execution of the threads will then continue in a similar fashion, executing the blocks in the following order: $t6$, $t7$, $t8$, and finally $t9$.

- I is *present* in the previous instant: Considering that the preemption condition is satisfied, the execution will be different. Like the previous case, execution will similar until PAR-3 starts. Here, the term $t3$ will complete and then the execution of the *checkabort* term will return a value of $k > 1$. Then, by the rule 18, the execution of the left branch of the first thread will pause until the execution of the term $t4$ in the right branch is completed. This will be followed by the execution of the *checkabort* statement in the right branch. This *checkabort* will also return a value of $k > 1$ that is identical to the value returned by the left branch. Hence, by the rule 19 PAR-3 will also return the same value. Now, by rule 18, execution of both thread 1 will pause and the execution of thread 2 will continue until the matching *checkabort* statement. Now PAR-1 will terminate with a value of k returned by both threads. When this happens, the enclosing weak abort statement will transfer execution to a term $t9$ by rule 24.

Definition 1 *The reaction of a program in an instant will be denoted as* $E : t \overset{1}{\underset{I}{\rightsquigarrow}} E' : t'$ *if there exits a sequence of transitions such that* $E : t \xrightarrow[I]{k_1} t_1(E_1) \cdots E_n : t_n \xrightarrow[I]{k} E' : t'$ *and* $k = 1$.

We now characterize PRET-C programs using the following two theorems. We start by defining *reactivity* and *determinism* in a standard way [25].

Definition 2 *A PRET-C program is reactive if, for any statement* t *and data set* E*, there exists at least one* ***reaction*** *for the set of inputs* I*, i.e., the program doesn't deadlock in any state given some valid inputs.*

Definition 3 *A program is deterministic if, for any statement* t *and data set* E*, there exists at most one* ***reaction*** *for the set of inputs* I*.*

Theorem 1 *All valid PRET-C programs are reactive, i.e.,* $\forall t$, $\forall E$, $\forall I$*, there exists* t' *and* E' *such that* $E : t \overset{1}{\underset{I}{\rightsquigarrow}} E' : t'$.

Theorem 2 *All valid PRET-C programs are deterministic, i.e.,* $\forall t$, $\forall E$, $\forall I$ *such that* $E : t \overset{k}{\underset{I}{\rightsquigarrow}} E' : t'$ *and* $E : t \overset{k'}{\underset{I}{\rightsquigarrow}} E" : t''$*, then* $t' = t''$, $E' = E''$, *and* $k = k'$.

The proof of both theorems is based on structural induction on t.

4 ARPRET Architecture

This section presents the hardware extension to a GPP MicroBlaze (MB) [24] in order to achieve temporal predictability. We designed the PRET-C language to enable the design of PRET machines by simple customizations of GPPs. The changes needed to execute PRET-C predictably concern the support for concurrency and preemption. If concurrency is implemented purely in software, the overhead of scheduling will be proportional to the number of parallel threads. Indeed, at each `EOT`, the scheduler has to select the next thread based on the status of threads and the preemption contexts. Doing this in software will consume significant numbers of clock cycles (compared to a hardware implementation), thus reducing the overall throughput. For this reason, we do the scheduling in hardware, with a custom made Predictable Functional Unit (PFU). Figure 6 shows the basic setup of an ARPRET platform consisting of a MB soft-core processor that is connected to a PFU.

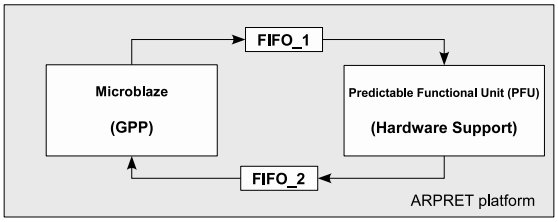


Figure 6: ARPRET platform consisting of MicroBlaze GPP and a PFU.

MB is a customizable RISC based soft-core processor, optimized for implementation on Xilinx FPGA. To guarantee predictability, some of its speculative features such as instruction and data caches were disabled. None of the features from the Memory Management Unit were used and no parallel shifters or floating point units were employed. We used five stages in the pipeline with the branching delay slot feature disabled.

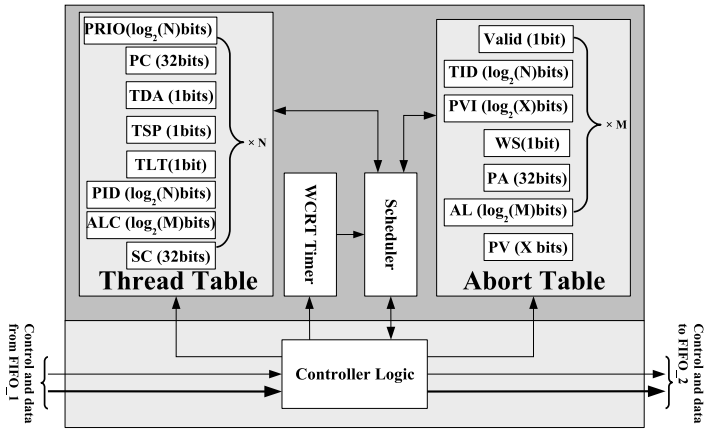


Figure 7: Predictable Functional Unit (PFU).

The PFU stores multiple thread contexts and schedules threads. For each thread, a thread table stores its Program Counter (PC) as a 32-bit value, its status (dead or alive, called TDA), its suspended status (TSP), its local tick status (TLT), and its priority (TP). Depending on the three thread status, the scheduler issues the next program counter when requested. Abort contexts are also maintained in an abort table for dealing with preemption.

MB acts as the master by initiating thread creation, termination, and suspension (this is different from Esterel suspend; in PRET-C, any thread that spawns child threads is suspended). The PFU stores the context of each thread in the thread table and monitors the progress of threads as they execute on the MB. When a given thread completes an `EOT` instruction on the MB, it sends appropriate control information to the TCB using $FIFO_1$. In response to this, the PFU sets the local tick bit (LTL) for this thread to 1, and then invokes the scheduler. The scheduler then selects the next highest priority thread for execution by retrieving its PC value from the thread table and sending it to MB using $FIFO_2$. Moreover, when all participating threads have completed their local ticks, the PFU waits for the tick length to expire. Whenever it completes a local tick, the MB blocks to wait for the next PC value from the PFU. It also waits when all ticks have completed their local ticks but the global tick hasn't happened. The tick length is decided by static WCRT analysis of a PRET-C program, as detailed in the report [19]. We next present in detail how the tightly coupled connection between the MB and the PFU is created in ARPRET.

Communication between MB and the PFU is done by using the Fast Simplex Link (FSL) interface [24] provided by Xilinx. This communication is done by the use of hardware FIFOs. FSL closely couples MB with the PFU using two FIFOs, called $FIFO_1$ and $FIFO_2$ respectively, to provide deterministic and predictable communication. Communication with FIFOs requires exchange of some common control signals such as the clock, reset, buffer status (FULL/EMPTY), read, write, and also data such as the PC value.

Function (ID)	Number of reads	Number of writes
SPAWN (10)	1	0
EOT (12)	1	1
SUSPEND (14)	1	1

Table 3: Simple look up table is used by the PFU to decode the data from $FIFO_1$.

The communication between the MB (master) and PFU (slave) is triggered when the MB executes instructions such as `PAR(T1,T2)` and `EOT`. The PFU contains a Controller Logic that refers to a look up table (LUT) as shown in Table 3 to decode the data from $FIFO_1$. For example, to spawn a thread, the MB writes a value of 10 followed by the start address of the thread into $FIFO_1$. Controller logic decodes 10 as SPAWN. Then, and by referring to the number of reads in Table 3, it fetches one more data element from $FIFO_1$ and stores it as the PC of the new thread. Also, in response to this SPAWN, other status bits such as the thread status and the suspended bits are altered appropriately.

Also, the TLT bit is set to 0 to indicate that the local tick for the thread is not yet reached.

5 Comparison with Esterel and Reactive C

This section seeks to illustrate the language through a set of examples and compares it with Esterel [4], ReactiveC [6] (denoted RC) and Reactive Shared Variables [7]. We are not comparing with the ECL language [13] since it is almost identical to Esterel semantically. We start by providing a qualitative comparison of the languages as shown in Table 4.

Criteria	Esterel	RC	PRET-C
Commutativity of ||	yes	no	no
Communication	signals	signals and variables	variables
Instantaneous Broadcast	yes	yes/no	no
Signals/Shared-variable values in an instant	single	multiple	multiple
Types of aborts	4	4	2
Types of suspend	4	4	0
Traps	yes	yes	no
Non-Causal Programs	possible	possible	not possible
Dynamic Processes	no	yes	no
Compilation	complex	complex	macro-expansion

Table 4: Qualitative Comparison with Esterel and RC

We now compare the three languages based on the criteria listed in 4.

1. Nature of Parallel: In Esterel, a parallel program is transformed to a sequential program during compilation. The compiler interleaves the execution of threads such that all producers of signals are scheduled before the consumers in each instant. Hence, it is possible to execute the same set of threads with different execution orders in different instants so as to respect the producer consumer dependencies of signals (see Figure 8). Unlike this, both in RC and PRET-C all threads execute based on a fixed static order that is repeated in every instant. Hence, the Esterel parallel operator is commutative while those in RC and PRET-C are not. While RC and PRET-C have similar approach to implementation of the parallel operator, the behavior of RC and PRET-C programs are quite different due to the difference in the way threads communicate. This is elaborated below.

2. Communication: Both in Esterel and RC, the main means of communication between threads is through a set of signals. In Esterel a signal is either present (true) or absent (false) in every instant. A signal may carry an additional value. Moreover, every signal can have only one single value in any instant (thus there can't be more than one emitter for a

```
[
  emit A(0);
  pause;
  emit A(?B+1)
||
  emit B(?A);
  pause;
  emit B(7)
]
```

```
PAR(T1, T2);
...
void T1(){
  A=0;
  EOT;
  A=B+1;
}

void T2(){
  B=A;
  EOT;
  B=7;
}
```

Esterel	A	0	8
	B	0	7
PRET-C	A	0	1
	B	0	7

Figure 8: Parallelism in Esterel and PRET-C

signal in any instant). To allow for multiple emitters of the same signal, combined-valued signals are supported where the multiple emissions must be combined using a commutative and associative operator. RC relaxes these restrictions by allowing multiple value emissions of the same signal (without a combine operator) where the last emitted value overrides the previous emissions (RC also supports combined-valued signals). It also allows the emitted signal to be reset within an instant using the reset statement. This is not possible in Esterel. One problem with RC is that signal checking is done dynamically and once a signal has been read, if a future emission happens, then a *run-time exception* is raised. This approach can lead to unpredictable run-time behaviour which is undesirable for programming PRET machines.

In contract to both Esterel and RC, all communication between threads are through variables (reactive and internal). Reactive input variables behave exactly like input signals of Esterel. These are read from the environment in the beginning of an instant and their value doesn't change during the entire instant. Reactive output variables and all other internal shared variables behave quite differently from both Esterel and RC signals. A reactive output variable can have multiple values within an instant due to multiple writers writing into this variable. Only the final value is emitted into the environment (this is similar to the RC approach of allowing multiple writers for signals). Unlike both Esterel and RC, all threads get unrestricted access to all shared variables, (see Figure 9). Since all threads execute in a fixed order and all reads and write are atomic, the program remains deterministic. RC also allows restricted access to shared variables where data-coherency requirement needs to be enforced by the programmer.

A variant of RC, called Reactive Shared Variable based Systems [7] uses only shared variables for thread communication like PRET-C. However, the view of determinism in this language is stricter. It demands that all readers must read the same values for data to be coherent. Hence, it introduces two phases in every instant. During the first phase, all writers of data are scheduled and in the second phase the readers are allowed to

```
1   PAR (T1, T2);
2   ...
3   void T1(){
4     while(1){
5       x=I+1;
6       if (x>MAX) x=MAX;
7       EOT;
8       x=I-1;
9       if(x<MIN) x=MIN;
10    }
11  }
```

```
11  void T2(){
12    abort
13      while(1){
14        x=2*x;
15      }
16    when pre(DONE);
17    x=x/2;
18  }
```

Figure 9: Multiple reads and writes to a shared variable in PRET-C

```
1 [
2   emit A;
3 ||
4   present A then emit B end
5 ||
6   abort
7     halt
8   when not A
9 ]
```

```
1 PAR(T1, T2, T3);
2 ...
3 void T1(){
4   a=1;
5 }
6
7 void T2(){
8   if (a==1) b=1;
9 }
10
11 void T3(){
12   abort
13     while(1){EOT}
14   when (pre(a==1))
15 }
```

Figure 10: Instantaneous broadcast in Esterel and PRET-C

read. Since we are interested in deterministic input-output behaviour (see definition of Determinism in Section 3), we take a less restrictive view of data-coherency and allow readers to read different values of the same variable as long as the overall program behaviour remains deterministic.

3. Instantaneous broadcast: In Esterel, communication between threads is through synchronous broadcast. Hence, once a signal is emitted, all other threads can test this value instantaneously. This is not directly possible in RC as the parallel operator is mapped to a fixed sequence. RC introduces the suspend operator to mimic synchronous broadcast behaviour through insertions of suspend. In PRET-C, we can't support the notion of instantaneous broadcast.

4. Preemption Support: In both Esterel and RC, four types of aborts (strong, weak, strong immediate and weak immediate) are possible. Also, four types of suspensions are possible. In PRET-C we only support two types of aborts that correspond to the strong immediate and the weak immediate aborts in RC/Esterel. We take this view to simplify the language and avoid the need for the distinction between *surface (behaviour in the first*

```
[
  present A then emit B end
||
  present B then emit A end
]
```

```
PAR(T1, T2);
...
void T1(){
  if (a==1) b=1;
}

void T2(){
  if (b==1) a==1;
}
```

Figure 11: Causality cycle in Esterel. In PRET-C, no information flows from T2 to T1.

instant) and depth behaviour (behaviour in all subsequent instants) of a program. Another distinction is that all preemptions in PRET-C are based on the evaluation of the condition in the previous instant. This is needed since variable values can change during an instant and by referring to the previous instant preemption always happen based on stead-state values. See Figure 10 for an example.

5. Causality: Esterel/RC can have non-causal [2] composition of threads due to feedback cycles. These must be rejected by the compiler and is the associated analysis causes large compiler overheads. PRET-C programs are causal by construction due to the semantics of our parallel operator. Hence, compilation of PRET-C programs is much simpler (only based on macro-expansion). See Figure 11 for an example of how there is a lack of information flow from T2 to T1.

6. Dynamic Process Support: RC is the most general language that allows dynamic creation of processes, which is not possible in both Esterel and PRET-C.

In summary, PRET-C is different from Esterel due to the nature of its parallel operator and also because of unrestricted access to shared variables across all threads. Another key difference is that we have no notion of surface and depth behaviours, no traps and suspends. Also, PRET-C programs are causal by construction. Due to the lack of synchronous broadcast communication, there is no need for scheduling threads. Hence, code can be generated from PRET-C through simple macro-expansion.

When comparing to RC, PRET-C inherits some semantic similarity with respect to the parallel operator. However, due to differences in data-coherency requirements, PRET-C allows unrestricted access to shared variables unlike RC. This makes programming task easier in PRET-C. In general, PRET-C, being a simpler language, is less expressive in comparison to RC and Esterel. For example, it can't directly express synchronous broadcast. Also, instantaneous dialogs [6], possible in Esterel is infeasible in PRET-C.

One of the key difference between PRET-C and earlier synchronous languages is our view of time. In both Esterel and RC, the time expressed in a program is purely logical. In PRET-C, a programmer is free to mix both logical time with physical time. Using our tools, the WCRT analyzer [19], the

Example	CEC			V5A			V5N			PRET-C	
	B	A	W	B	A	W	B	A	W	$WCRT_{max}$	WCRT
ABRO	60	78	109	83	185	266	367	403	438	89	89
Channel Protocol	137	232	313	86	271	432	967	1099	1149	174	152
Reactor Control	89	112	144	144	189	311	633	677	714	121	118
Producer Consumer	275	408	417	397	524	596	608	742	763	118	99
Smokers	33	552	1063	80	723	1256	743	1186	1603	531	449
Robot Sonar	275	408	417	397	524	596	608	742	763	419	346
Average	145	298	410	198	402	576	654	808	905	242	208

Table 5: Quantitative comparison of execution time with Esterel

programmer can then view the mapping of the logical time to physical time automatically. The proposed analysis is based on model checking and returns the tight WCRT value for a given program. Hence, this approach is an excellent choice for PRET implementation as throughput is not wasted through overestimated WCRT value of ticks. In effect, our mapping is identical to hardware compilers for Esterel. More details about WCRT analysis of PRET-C programs are available in [19].

Example	CEC (Bytes)	PRET-C (Bytes)
ABRO	21436	12340
Channel Protocol	22784	17628
Reactor Control	21800	12716
Producer Consumer	26926	17060
Smokers	22432	12716
Robot Sonar	22388	15796
Average	22959	14709

Table 6: Code size comparison between CEC and PRET-C.

6 Benchmarks and results

PRET-C is a new language and hence there are no existing benchmarks. We created three sample programs called *Smokers* [1], *Robot Sonar*, and *Producer Consumer* presented in Section 2.1. The *Robot Sonar* example was developed considering the application domain for PRET-C. These three PRET-C programs were also modeled in Esterel. To preserve behavioral equivalence, we replaced all non-immediate preemptions in the benchmarks in Esterel with their immediate counterparts. Since we are comparing with Esterel, we also modeled three examples from the Estbench [10] suite in PRET-C while keeping their behavior identical (*ABRO*, *Channel Protocol*, and *Reactor Control*).

6.1 Benchmarking

The benchmarking process was carried out as follows. Firstly, we generated code on ARPRET such that the generated code executed without any of the speculative features available on the host processor (MB), such as branch prediction. For Esterel, the code was generated on MB with branch prediction enabled. Since the benchmarks currently are small, both Esterel and PRET-C code fit on the on-chip program memory. Hence, the behavior of caches have not been taken into account.

For Esterel, we generated code using a range of compilers. We then compared the generated code size (in bytes) and execution time (in MB clock cycles) with that of PRET-C. For execution time comparison, we used random test vectors and measured the execution time over one million reactions. The best case is the minimum measured time over the samples, the worst case is the maximum of the measured values, and the average is obtained by averaging all samples.

The results of these experiments is presented in Table 5. The first column indicates the benchmarks used. The next three columns of the table show the generated execution time results for Esterel, respectively with the compilers CEC, Esterel-V5 (automaton code), and Esterel-V5 (netlist code). Each column is further split into three sub-columns showing the best case (B), average case (A), and worst case (W) execution times. The next column shows the execution time results of PRET-C. For PRET-C programs, we first determined the worst case tick length (WCRT), by a static analysis method developed by us [19]. We then set the processor's tick length to this WCRT value. We can also statically determine the maximum WCRT value of a program by adding the maximum tick lengths of all participating threads (called WCRT_{max}). For simple programs that have a small number of threads and don't have any data dependency between threads (such as ABRO), the WCRT_{max} value is identical to the value obtained by static analysis (WCRT column). However, for programs that have a large number of threads and good computation to communication ratio such as the Robot Sonar example, the static analysis method produced tighter results. This is because the method of [19] considered both data dependency and state-based contextual information to remove redundant paths for tighter analysis.

The WCRT value of PRET-C is systematically better than the worst measured execution time obtained with the three Esterel compilers. In most cases, the WCRT value of PRET-C was also better than the average measured execution time obtained with the three Esterel compilers. This is because PRET-C is executed very efficiently using ARPRET where the concurrency and preemption are implemented in hardware. Also, PRET-C doesn't have any compilation overhead (as it requires only macro-expansion) to generate C code. It may be noted, however, that we haven't taken the cache behavior into account because all the programs fit in the on-chip memory. For larger programs requiring the cache, these results will probably be different. This is a limitation and we are working on this issue to develop predictable memory hierarchy for ARPRET.

Table 6 compares the code size from the most compact Esterel compiler (CEC) with that of PRET-C. The code for PRET-C was on the average about 35% more compact.

We next present the results of the hardware resource usage on the FPGA device. The hardware resources in terms of Slices and Look Up Tables (LUT) are shown by Figure 12. Slices are logical blocks providing functionality such as arithmetic, ROM functions, storing, and shifting data. They contain LUTs, storage elements, and multiplexers. Four-input look-up tables are used by FPGA function generators for implementing any arbitrarily defined four-input Boolean function [24]. From Figure 12, we can see that the hardware resource consumption of ARPRET is linearly proportional to the number of threads. This is due to the fact that ARPRET mostly stores thread contexts, and only minimal datapath is required by the scheduler.

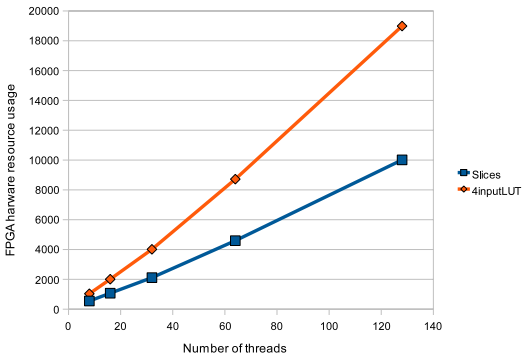


Figure 12: Number of threads versus hardware consumption in terms of LUTs.

6.2 Comparison with the Berkeley-Columbia PRET approach

Criteria	Berkeley-Columbia approach	Auckland approach
Type of design	Tailored processor for PRET	Customized GPP for PRET
Notion of time	Physical (`deadi` instruction)	Logical (`EOT` and WCRT analysis)
Mutual exclusion	Through time interleaved access	By construction
Memory hierarchy	Handled through scratch pad memories	No
Prototyping	Emulation using a SystemC model	Full implementation on FPGA
Input language	Concurrent C without formal semantics	Synchronous extension to C
Support for preemption	No	`abort when pre` statement

Table 7: Qualitative comparison between Auckland and Berkeley-Columbia approaches.

A qualitative comparison between our approach and the Berkeley-Columbia approach is summarized in Table 7. The main goals of our approach and Berkeley's are to provide predictability and simplify WCET analysis, while maintaining throughput. They both extend C with multithreading, and use hardware support to maintain efficiency.

Berkeley's thread interleaved pipeline execution model requires a minimum of six threads to work efficiently. For complex programs, calculating the values for the `deadi` instructions to guarantee time interleaved access of shared variables can be very tricky. Their architecture model is built on SystemC around

a SPARC GPP and includes a scratchpad memory, a six-stage pipeline, and a memory wheel. This model substantially modifies the SPARC GPP.

In comparison, ARPRET executes threads sequentially between logical instants and there is no need to stall any pipeline stages. Having a notion of tick abstracts away time while also simplifying WCRT analysis [19]. Also, calculating the length of the tick is far simpler than calculating the values for `deadi` instructions, and is performed automatically by our compiler instead of manually by the programmer. Our hardware architecture is also simpler than Berkeley-Columbia's since it can be used with any customizable processor. This, in our opinion, is a more flexible approach for the design of PRET machines. However, the Berkeley-Columbia research is more mature than the Auckland approach and they have made progress in several directions such as memory hierarchy issues and the development of larger applications such as a video game controller.

7 Conclusions and future work

Precision Timed (PRET) architectures are a recent attempt to design processors that guarantee predictable execution of code without sacrificing throughput. Researchers from Berkeley and Columbia proposed a tailored processor with a thread interleaved pipeline and a low-level instruction to introduce precise timing in C code [15]. In contrast, this paper proposes the customization of embedded soft-core processors to design PRET architectures with minimal hardware requirements. We also propose a new language for programming PRET machines, called PRET-C, by simple synchronous extensions to the C language. PRET-C has constructs for expressing logical time, preemption, and concurrency. Concurrent threads communicate using the shared memory model (regular C variables) and communication is thread-safe by construction. We have designed a new PRET machine, called ARPRET, by customizing the MicroBlaze soft-core processor. We have benchmarked the proposed design by comparing the execution of PRET-C on ARPRET with the execution of Esterel on its speculative counterpart (MicroBlaze). Benchmarking results reveal that the proposed approach achieves predictable execution without sacrificing throughput.

In the future, we shall extend our design to be able to deal with memory hierarchy issues for ARPRET to execute large embedded applications predictably and efficiently. We shall also develop tools that can highlight the timing behavior of code to the programmer statically. This will allow the programmer to customize the code depending on the application requirements. Another area to be examined is the execution of PRET-C on multicore architectures. One possible extension could be a GALS-based language that is amenable to multicore execution.

References

[1] G. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, Jan 2003.

[3] G. Berry. Constructive semantics of Esterel: From theory to practice (abstract). In *AMAST '96: Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology*, page 225, London, UK, 1996. Springer-Verlag.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics and implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[5] M. Boldt, C. Traulsen, and R. von Hanxleden. Worst case reaction time analysis of concurrent reactive programs. *Electronic Notes in Theoretical Computer Science*, 203(4):65–79, June 2008.

[6] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, April 1991.

[7] F. Boussinot. Reactive shared variables based systems. Technical Report 1849, INRIA, 1993.

[8] F. Boussinot and R. de Simone. The SL synchronous language. *IEEE Trans. Softw. Eng.*, 22(4):256–266, 1996.

[9] B. Cogswell and Z. Segall. MACS: A predictable architecture for real time systems. In *Real-Time Systems Symposium*. IEEE CS Press, 1991.

[10] S. Edwards. Estbench Esterel benchmark suite. http://www1.cs.columbia.edu/ sedwards/software/estbench-1.0.tar.gz.

[11] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *Proceedings of the 44th annual conference on Design automation*, pages 264–265. SESSION: Wild and crazy ideas (WACI), June 2007.

[12] S. A. Edwards and J. Zeng. Code generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems*, 2007. Article ID 52651.

[13] L. Lavagno and E. Sentovich. ECL: A specification environment for system-level design. In *Proceedings of Design Automation Conference (DAC)*, New Orleans, USA, June 1999.

[14] X. Li, M. Boldt, and R. von Hanxleden. Mapping Esterel onto a multi-threaded embedded processor. *SIGARCH Comput. Archit. News*, 34(5):303–314, 2006.

[15] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *In Proceedings of International Conference on Compilers, Architecture, and Synthesis from Embedded Systems*, October 2008.

[16] H. D. Patel, B. Lickly, B. Burgers, and E. A. Lee. A timing requirements-aware scratchpad memory allocation scheme for a precision timed architecture. Technical Report UCB/EECS-2008-115, University of California, Berkeley, Sept. 2008.

[17] G. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[18] P. Roop, Z. Salcic, and M. Dayaratne. Towards direct execution of Esterel programs on reactive processors. In *4th ACM International Conference on Embedded Software (EMSOFT 04)*, 2004.

[19] P. S. Roop, S. Andalam, R. von Hanxleden, S. Yuan, and C. Traulsen. Tight WCRT analysis of synchronous C programs. Technical Report 0912, Christian-Albrechts-Universität Kiel, Department of Computer Science, May 2009. www.ece.auckland.ac.nz/∼roop/pub/2009/roop-report09.pdf.

[20] Z. Salcic, P. Roop, M. Biglari-Abhari, and A. Bigdeli. Reflix: A processor core for reactive embedded applications. In *12th International Conference on Filed Programmable Logic and Applications (FPL-02)*, 2002.

[21] M. Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, 2009. Article ID 758480.

[22] F. Vahid and T. Givargis. *Embedded System Design*. John Wiley and Sons, 2002.

[23] R. von Hanxleden. SyncCharts in C. Technical Report 0910, Christian-Albrechts-Universität Kiel, Department of Computer Science, May 2008.

[24] Xilinx. *MicroBlaze Processor Reference Guide*, 2008.

[25] L.-H. Yoong, P. S. Roop, V. Vyatkin, and Z. Salcic. A synchronous approach for IEC-61499 function block implementation. *IEEE Transactions on Computers*, 2009 (to appear).

[26] S. Yuan, S. Andalam, L.-H. Yoong, P. S. Roop, and Z. Salcic. STARPro: A new multithreaded direct execution platform for Esterel. In *Model-driven High-level Programming of Embedded Systems (SLA++P'08)*, 2008.

Contents

List of Figures

List of Tables

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399